Reconfigurable Applications Using GCMScript

Matías Ibañez, Cristian Ruz, Ludovic Henrio, Javier Bustos-Jiménez

Reconfigurable Applications Using GCMScript

Matías Ibañez,Cristian Ruz, Ludovic Henrio,Javier Bustos-Jiménez

Abstract

Nowadays, applications are commonly deployed in distributed environments using multiple computational resources from elastic cloud infrastructures. Environmental conditions, however, may vary through different providers or even at runtime and the applications must be able to quickly adapt to new conditions. Component based applications plus autonomic computing appear as flexible solutions to make applications reconfigurable and more adaptable. Programming of autonomic behavior, however, is not an easy task. In this work we use a component based framework to build reconfigurable and distributed applications, plus a scripting language to facilitate the programming of autonomic behavior. We show the effectivity of our approach using a distributed master-worker application that is able to self-adjust its load.

Index Terms

Autonomic computing, reconfiguration, cloud computing

Reconfigurable Applications Using GCMScript

1 Introduction

Nowadays, applications are usually deployed in distributed environments like grid or cloud infrastructures, where it is easy to acquire elastic computational resources over which different components of an application can be deployed and run in parallel. Such distributed architectures are affected by environmental conditions as network traffic, request rate, cost, or security requirements, which may vary for different cloud providers and affect the application at deployment time or even at runtime, sometimes unexpectedly. Cloud-based applications need to be dynamically reconfigured to adapt to the new conditions in a short time, in order to meet a required quality of service and avoid lengthy downtimes.

The need for rapid reactions and the complex management requirements that may govern different cloud services reach a level that discards the intervention of human administrators. The answer to these problems has been the introduction of autonomic behavior on the applications, so that the application can manage itself.

The distributed nature of the applications makes impractical the existence of a centralized knowledge of the whole application architecture. In this context, component based software development appears as a suitable model to build applications through the composition of loosely coupled entities that can be deployed and run in different virtual resources. Component based applications are also naturally designed to make them reconfigurable and adaptable through simple modification of bindings, and several component models have been proposed and used in this regard. Component-based systems, such as those built using Fractal [1], or GCM[2], are specially created to facilitate reusability and they are naturally prepared to adapt to their execution context. Adaptation, however, is not always easy to achieve, even less to program, and some developments (such as FScript [3]) are specifically designed for a less error-prone reconfiguration.

In this work we show how it is possible to program autonomic reconfigurations of distributed component-based applications by using the GCMScript language. GCMScript has been designed as an extension of FScript allowing distributed execution of reconfiguration actions through the collaboration of several GCMScript interpreters embedded in GCM applications, that may run

in one or several cloud environments. GCMScript actions, plus the introduction of autonomic control loops inside GCM applications allow us to build effective autonomic behavior into an application. This autonomic behavior can be programmed to dynamically modify the architecture of the application, to deploy or remove components using the elastic nature of the infrastructure, or to modify runtime parameters in order to scale and provide a better QoS management.

We present an example GCM application over which self-optimizing behavior can be introduced using GCMScript. In this application, self-optimization can be triggered in two ways, either (1) through an autonomic control loop that continually evaluates the application performance and modifies its parameters in order to improve itself; or (2) through the autonomic adaptation triggered by a modification in the environment.

2 GCM AND GCMSCRIPT

Our work is shown in the context of the GCM component model and the GCMScript language for programming distributed reconfigurations.

2.1 GCM

The Grid Component Model, GCM[2], is a hierarchical component model adapted from Fractal[1] for large-scale distributed computing. The structure of a GCM component assembly and the terminology used are shown in Figure 1. Services are offered through *server interfaces*, and required services should be bound to *client interfaces*. GCM components provide collective interfaces, that may be one-to-many (multicast), many-to-one (gathercast), and many-to-many (MxN) interfaces. One-to-many interfaces transform a single invocation into many invocations, potentially distributing the invocation parameters. Many-to-one interfaces wait for several invocations and transform them into a single one, potentially aggregating the parameters received. MxN interfaces feature both aspects. Another architectural addition in the GCM is the possibility of structuring the non-functional (NF) part, i.e. the *membrane*, through NF components. NF components share the same features as regular functional (F) components: they can be assembled, composed and introspected. The main advantage of using such a membrane are (1) the separation of NF concerns from the functional part, and (2) the possibility of designing and implementing complex and adaptable NF behaviors.

2.2 GCM/ProActive

GCM/ProActive[2] is the implementation of the GCM component model based on active objects. In GCM/ProActive, each component is implemented by an active object, which is an object with

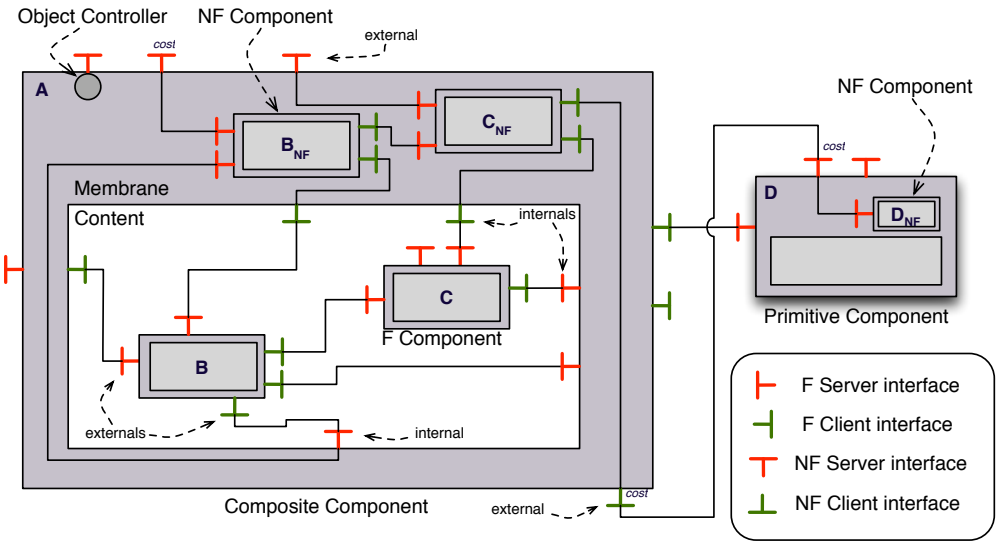

Fig. 1. Grid Component Model (GCM) notation

a single thread of execution and a service queue. Active objects may have their own execution policy and implement asynchronous requests by using *futures*. One of the main advantages of using active objects to implement components is their adaptation to distribution. Active objects provide a natural way to build loosely coupled components, i.e. components responsible for their own state and execution, and only communicating via asynchronous messages. The GCM/ProActive implementation of the membrane can also be seen in Figure 1. The figure shows two components B_{NF} and C_{NF} in the membrane of a composite component A, and one component D_{NF} in the membrane of primitive component D.

The separation of concerns between F and NF parts allows us to design each behavior in separate ways and to delay the integration of both aspects until deployment time. Even then, after the component has been instantiated, it is possible, through introspection, to analyze and dynamically modify both the F or NF components without affecting the rest of the architecture. This decoupling permits the management concerns to be designed by an expert that does not need to know about the functional behavior, and, conversely, the programmer of the F part does not need to focus on the NF concerns.

Concerns like composition, bindings, monitoring, reconfiguration, security policies enforced through different cloud providers, load balancing by elastically using more or less virtual instances, and in general self-management activities either of the application or related to the cloud infrastructure, can be designed without having to know the details of the functional part,

focusing only on the component management. However, at execution time, both F and NF parts need to interact. In particular, a NF concern related to self-management may need to know the composition or to observe the behavior of the F part, and it may need to modify some attribute or to carry on an action that affects the F part.

Self-management behaviors may require a complex implementation, and it becomes reasonable to separate the application control in different components. By using a "component-ized" membrane we are able to implement complex tasks using components that, similarly to the functional part, can be assembled, composed, and replaced.

As an example, in Figure 1, components A (including subcomponents B and C), and D may run on different cloud providers with different pricing models. The cloud provider of A charges a fixed amount per day, while the cloud provider of D charges depending on incoming traffic on D. In order to compute a unified *cost* metric, A can use a NF component C_{NF} that keeps track of the uptime days of A, while D_{NF} keeps track of the requests received by D. A component B_{NF} can be programmed to query both C_{NF} and D_{NF} (using an inter-cloud NF binding) and determine the cost of running the whole application. This metric is exposed to a NF interface for further composition. Also note that it is possible to modify the pricing model by changing only the NF components without modifying the F components. In Section 5 we use insertion of NF components to provide autonomic behavior in an application.

2.3 GCMScript and Reconfiguration Controller

In a similar way to Fractal, GCM components expose control interfaces that allow to modify, at runtime, the application structure. Using those interfaces it is possible to add or remove components or bindings at runtime, allowing to implement dynamic reconfigurations.

In order to ease the programming of reconfiguration procedures, we have designed the *GCMScript*[2] language, dedicated to reconfiguration of distributed components and, in particular, to the reconfiguration of GCM components. GCMScript is based on an extension of FScript [3], the reconfiguration scripting language used by Fractal components.

FScript is executed by a centralized interpreter located in a global component that has knowledge of all the architecture. GCM applications, however, do not have that restriction, and GCMScript is be able to interpret reconfiguration scripts in a distributed manner. In this way, composite components can be responsible for reconfiguring their inner components independently, allowing reconfiguration to take place in parallel and facilitating their execution

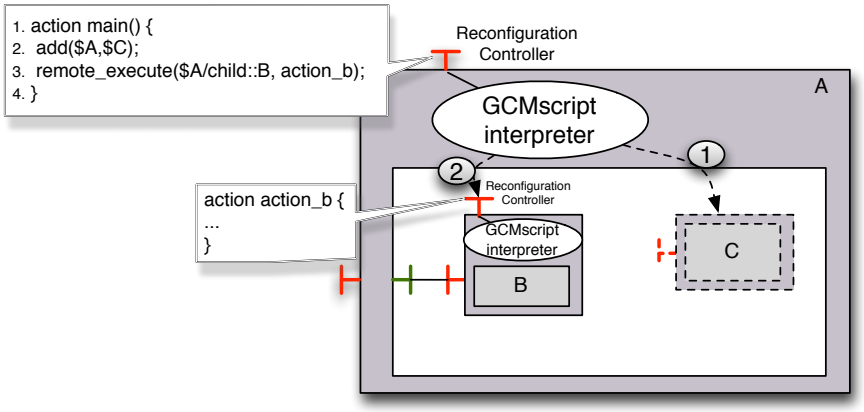

Fig. 2. GCMScript interpreter delegation

in large-scale infrastructures. Access to GCMScript engines is achieved through a controller interface called *Reconfiguration Controller*, as it is shown in Figure 2.

The *Reconfiguration Controller* embeds an instance of an FScript interpreter and provides methods `load` and `execute` for loading and triggering reconfiguration scripts and actions. In order to ensure consistency and to avoid unpredictable interplay between execution and reconfiguration actions of a component system, it is necessary to ensure that the functional behavior of a component is stopped while reconfiguration takes place. As we are dealing with an asynchronous setting, we rely in a protocol able to stop an asynchronous composite component together with all its subcomponents in a safe way [4]. The objective of such a protocol is to ease the design of safe adaptation procedures: when a subsystem is entirely stopped, it can go through a reconfiguration phase before being restarted.

3 Background and Related Work

In 2009, at the *Grids Meet Autonomic Computing* workshop, researchers focused on the key challenges in grid and cloud computing that autonomic computing techniques support [5]. In the workshop panel, researchers and practitioners observed that "*existing cloud computing infrastructure already supports some autonomic concepts, such as determining the number of virtual machines (VMs) to support, providing dynamically expandable storage, and migrating workloads across computational platforms*". In this scenario, the convergence of cloud computing with autonomic components seems to be natural and highly synergic.

One of our first attempts in autonomic computing for grid/cloud computing was the use of coupling contracts for Proactive's active objects (the base of GCM), with those contracts, autonomic behaviors such as dynamic load balancing were achieved [6].

Component models [7] aim to increase the code reusability, and adaptability of software, by introducing well-delimited software entities with clearly defined interfaces corresponding to either the offered services or those explicitly required to fulfill them. Component-based applications differentiate from other kind of applications by making the resulting software architecture explicit. As in the Proactive's contracts, the component architecture is described using an assembly language (ADL). Associated to an ADL, a factory instantiates all the components that constitute the application using a parser and component generators. This step deploys base components from their source implementation and binds component instances according to the dependencies defined in the ADL.

Hierarchical composition is crucial for easing the creation of large applications. By gluing together components to form another component that can be used itself in the composition, it is easier to design larger applications and also to use different deployment concerns over them. For example, deploying groups of components in a cloud provider by deploying a single composite components that contains them all.

Besides using directly an API, modifications over the application can also be expressed through scripting languages, associated to the ADL, as done in ArchJava [8] or built upon an API as FScript [3] for Fractal components. Allowed modifications can also be the result of some component-associated constraints as in [9] or reconfiguration rules (Event Condition Action rules). As in our case, most of them rely upon the *Monitor/Analyze/Planner/Execute* (MAPE) model for autonomic computing firstly presented by IBM [10], however our framework allows a reconfiguration to take place in a distributed manner instead of relying on a centralized control. To enable the dynamic reconfiguration of the control layer itself, this layer should be programmed yielding a "component-ized" membrane concept as presented in [11]. Besides Fractal/AOKell, the SOFA component model whose control part is built using micro-components [12] and Dynaco that uses full-fledged components [13], in GCM such flexibility is achieved by the fact that the control level is also made of components.

In the field of providing autonomic behavior to cloud applications, works like [14] and [15] use MAPE loops in the architecture of the application and provide a hierarchic organization of managers. Although they effectively collect the required metrics, our scheme allows to introduce the autonomic behavior as part of the application itself, instead of a separate architecture. Plus,

our componentized membrane model together with the scripting language allows us to program the autonomic behaviour in a more generic and less error prone manner.

4 AUTONOMIC COMPONENT SYSTEM

Our Autonomic Component System (ACS) is a framework that relies on NF components and GCMScript to facilitate the creation of GCM applications that contain autonomic behavior and can be deployed and applied over GCM applications running on cloud environments.

The design relies on a set of NF components based on the stages of the MAPE control loop, and a set of GCMScript actions that allow to activate or modify the behavior of these components at runtime. To this effect, one GCMScript interpreter is located in each GCM component, and accessed through a *Reconfiguration Controller*.

4.1 Configuration Elements

Following the MAPE autonomic control loop, four components are defined: *Monitor*, *Analyze*, *Planner* and *Execute*. In previous works we have developed GCM/ProActive autonomic applications following this scheme, however the insertion and configuration of those NF components is quite cumbersome and error-prone as it requires several calls to the component API, and additional precautions must be taken in order to stop and start membranes and components in the appropriate order. In this work we have provided a set of GCMScript actions that simplifies the construction of the self-adaptive behavior.

ACS aims to make the application compliant with certain high-level objectives. These objectives are defined by three configuration elements: *Metrics* in the *Monitor* component, *Rules* in the *Analyze* component, and *Plans* in the *Planner* component. In our implementation each one of these elements are Java classes, defined by the user as an extension of a provided parent class. Their instances operate together to define the self-adaptive behavior. ACS provides actions to configure the *Monitor*, *Analyze*, and *Planner* components using these elements. The *Execute* component, on the other side, has a fixed configuration embedding the GCMScript interpreter.

4.1.1 Metric

The *Metric* element is able to perform measurements over the application at runtime. The GCM/ProActive implementation generates a set of general purpose events at runtime, which are caught and stored by the *Monitor* component. A *Metric* that is loaded into the *Monitor* component can use the stored data to measure non-functional aspects, f.e. how many times a component

receives requests from other components through a server interface, or the average response time for a particular interface. Measurement can be done in two modes: manually by calling a method of the *Monitor* component that gets or computes the value for the *Metric*; or in an automatic way using event-subscription, so that whenever a GCM/ProActive event is caught, the metric value is updated and sent to all the subscribers.

Additionally, if a component A depends on the services of a component B (i.e. A has a functional binding to B), then the *Monitor* of component A has access (through a non-functional binding) to the metrics of component B. This property is transitive and allows more complex and varied metrics to be built into the application.

4.1.2 Rule

The *Rule* element represents an SLO (Service Level Objective) that must be enforced during runtime. A *Rule* relies in the value of a subset of *Metric*s that must be available in the *Monitor*, in order to verify the state of the execution and decide the state of the *Rule* compliance. In case the rule is not complied, an alarm is automatically triggered.

After a *Rule* has been added to the *Analysis* component, its state can be verified manually, by calling a method on the *Analysis* component, by setting up an update period, or automatically by subscribing to *Metric*s, so that the *Rule* is checked whenever the metric value changes.

4.1.3 Plan

The *Plan* element handles one or more *Rule* alarms and decides the necessary changes that must be applied to the application in order to restore the breached rule(s). In our implementation, a *Plan* element has access to the *Reconfiguration Controller* and, through it, it can execute any necessary GCMScript command.

*Plan*s are inserted into the *Planner* component. Execution of the *Plan* can be requested manually by invoking a method on the *Planner*, or automatically by subscribing to *Rule*s, so that whenever the rule triggers its alarm, the *plan* is executed.

4.2 GCMScript extension

In order to ease the building of reconfiguration actions, the elements described are explicitly incorporated into the GCMScript model. These extensions allow to browse, using FPath queries, the elements that are part of the reconfiguration architecture, and also to introspect the subscription relationships between reconfiguration elements.

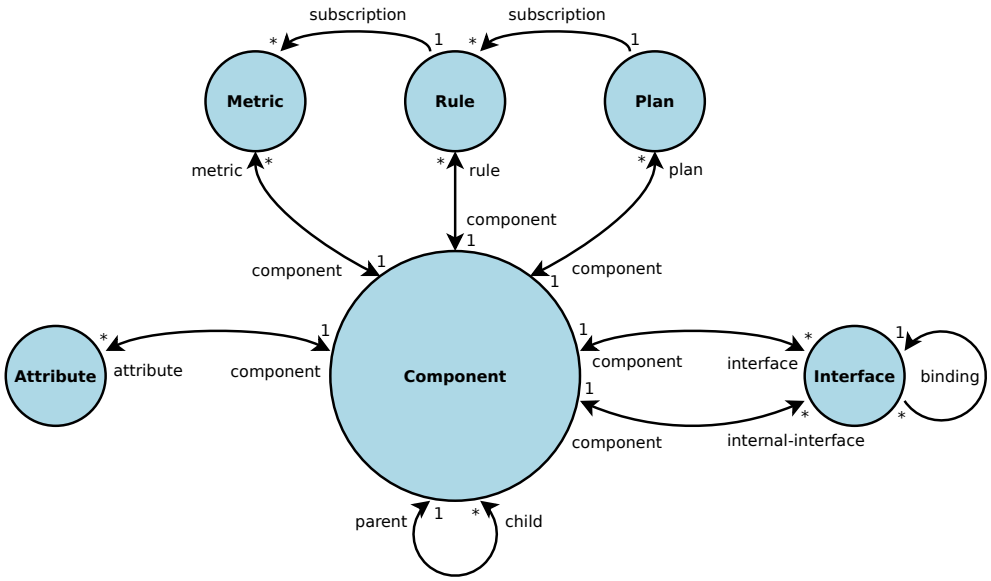

Fig. 3. The GCMScript Model with the extension for Metric, Rule and Plan

The goal is to manage the reconfiguration process using uniquely GCMScript commands, and keeping these concerns detached from the main application, easing the programming of the autonomic behavior. A node for each configuration element was added to the model as shown on Figure 3. Keeping them as separate nodes allows to easily reach them through FPath expressions, avoiding explicit API calls.

FPath queries use the model to navigate through the component architecture, where nodes are elements that can be queried and edge labels allow to navigate through the nodes. For example the expression `$c/child::d` returns a reference on component `d`, which is a sub-component of `c`; and `$c/metric::avgResponseTime` returns a reference to the metric called *avgResponseTime* that is located inside the *Monitor* component of `c`.

4.2.1 GCMScript procedures

The main procedures introduced to GCMScript are listed below.

- Add: an element is added to a component

```
add-metric($hostComponent, "fooName", "cl.example.FooMetric");
add-rule($ruleNode, "fooRule", "cl.examples.rules.MyRule");
add-plan($planNode, "fooPlan", "cl.examples.plans.MyPlan");
```

- Remove: an element is removed from a component.

```
remove-metric($hostComponent/metric::fooName);
```

```
remove-rule($hostComponent/rule::fooRule);
remove-plan($hostComponent/plan::fooPlan);
```

- Get property values: query the element state

```
value($myComponent/metric::myMetricId);
alarm($myComponent/rule::myRuleId);
```

- Subscription:

```
fooRule = $hostComponent/rule::myRuleId;
add-subscription($hostComponent/plan:fooPlanId, $fooRule);
```

4.2.2 GCMScript Console

FScript provides a console application that allows to interactively execute commands over an FScript interpreter. We have also extended this console in order to integrate it with the GCMScript model. Since every component embeds its own GCMScript engine inside the ReconfigurationController, the GCMScript Console consists of a user-component communication, instead of user-system communication. The console hides the notion of GCMScript engine and allows the user to browse through components until reaching the desired one, and send commands there. This interaction is shown in Figure 2.

5 USE CASE: SELF-BALANCED DISTRIBUTED BRUTE FORCE CRACKER

The following example shows how our proposed ACS can be used to build self-optimizing applications that are able to autonomically balance its workload. The application is a distributed brute-force cracker that uses a set of available machine resources, each one of them handled by a GCM/ProActive component, hosted in different virtual instances, where each one of them may be located in different clouds.

The application, shown in Figure 4, works by testing a set of N words, and computing its MD5 hash. The *Manager* component distributes the load by splitting the job in three subsets J_1, J_2, J_3, $|J_1| + |J_2| + |J_3| = N$ of words, and sends each one of them to a different *Solver*. Each *Solver* component is deployed on a different cloud resource and follows a master-slave model to provide local multi-core parallelism to the cracking process; inside, the *Master* component evenly splits the set of words among the available *Slave* components.

At the beginning of the execution there is no autonomic behavior enabled and the *Manager* distributes an equal amount of work for each *Solver*, so $|J_1| = |J_2| = |J_3|$. When the self-optimization capability is activated, the system evaluates the throughput of each *Solver*

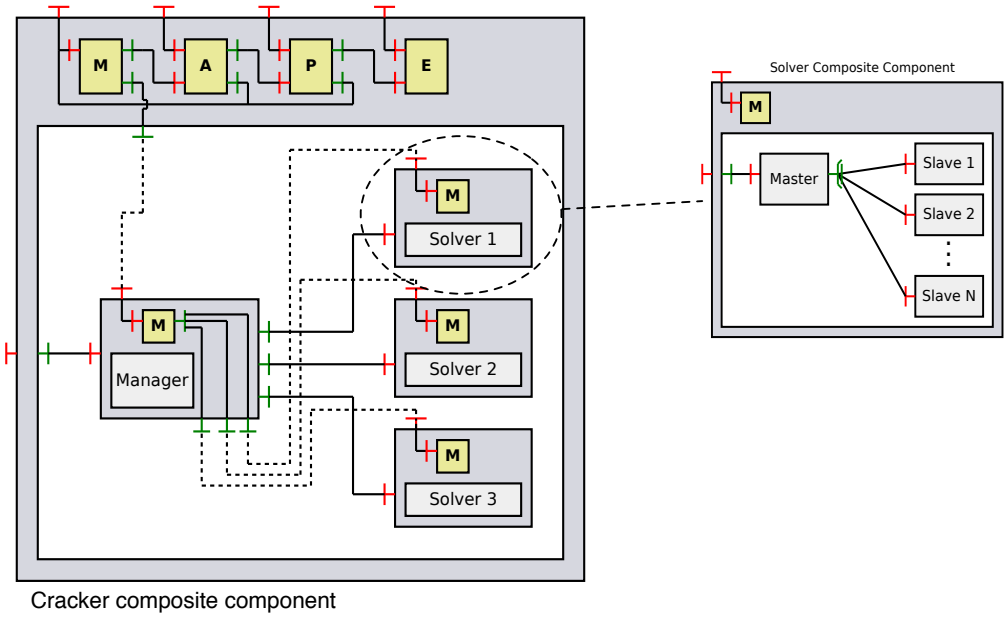

Fig. 4. The MD5-Hash Cracker component

component and tries to balance the load among the *Solvers*, so that they take similar amounts of time, thus avoiding the bottlenecks that happen when the answer is delayed by the slowest solver (note that there is no expropiation of tasks in this experiment).

The amount of work distributed to each solver can be manually changed at any moment using the *AttributeController* of the *Manager*. In our experiment, we will use this capability to introduce an autonomic behavior that chooses the optimal load distribution.

5.1 Setting up the reconfiguration elements

The setting up of the autonomic cracker application goes through the following steps:

1) Obtain computational resources for each component, and deploy a GCM/ProActive component in a different resource.
2) Add a *metric* called $avgRT_i$ on each Solver. This metric calculates the average response time for *solver* i, and will be used as a performance metric of the solver.
3) Add a *metric* called *optimalBal* on Cracker that calculates an aproppriate balance for the load given to the solvers. This metric is computed using the value of each $avgRT_i$.
4) Add a *rule balanceStatus* on the Cracker component that checks that *optimalBal* does not change by more than a tolerance d, and subscribe the rule to the event "receive a new cracking request".

5) Add a *plan* on the Cracker component that replaces the current load distribution on the *Manager*. Subscribe this plan to the rule *balanceStatus*.

5.2 Parameters and tests

The application is deployed in a cloud using 2 *Slave*s on each *Solver*. Both *Solvers* and the *Manager* are deployed on different instances. *Solver* 1 on a machine with 8 x 1200 Mhz cores, *Solver* 2 on a machine with 8 x 1600 Mhz cores, and *Solver* 3 on an instance with 8 x 2200 Mhz cores. The execution is divided in three phases:

1) **Phase 1. First 40 requests:** Cracker running without any autonomic behavior enabled. The default load balance $|J_1| = |J_2| = |J_3|$ is used for every request.
2) **Phase 2. Requests 41 to 80:** Self-optimization activated. From the 41st request, self-optimization elements are enabled.
3) **Phase 3. Requests 81 to last:** Environmental change. Before the 81st request, 4 additional *Slave*s are added to *Solver 1*, and 2 additional *Slave*s are added to *Solver 2*. These modifications are manually done using the GCMScript console.

5.3 Results

Figure 5 shows the load assigned to each *solver* at each phase of computation. Figure 6 shows the performance as measured by the metric *avgRT* for each *solver*. In this case, the Y-axis represents time, noticing that the performance of the whole system as it is seen by a client is $max\{S_i\}$. In fact, a user of this application is not interested in the individual times taken by each *Solver*. Even more, the user does not need to know the existence of several *Solver*s, or the distributed cloud nature of the application, even if more than one cloud provider are being used. The user only perceives the time the application, as a whole, takes to respond to the requests. There might exist, however, the concern that a modification that requires deployment of a new instance on multiple clouds will have an increased time to adjust than one deployed in a local cloud.

During Phase 1, equals amounts of work are sent to each solver. Since the machines have different hardware capabilities, this distribution (same load for each solver, according to Figure 5) is not optimal, as it is evidenced on the first part of Figure 6, during which *Solver 1* seems to be the slowest. During Phase 2, the self-balancing system is turned on, and the application autonomically determines that the current balance is far from *optimalBal*. Consequently, a new load distribution is computed. After the *plan* updates the new load distribution on the *Manager*,

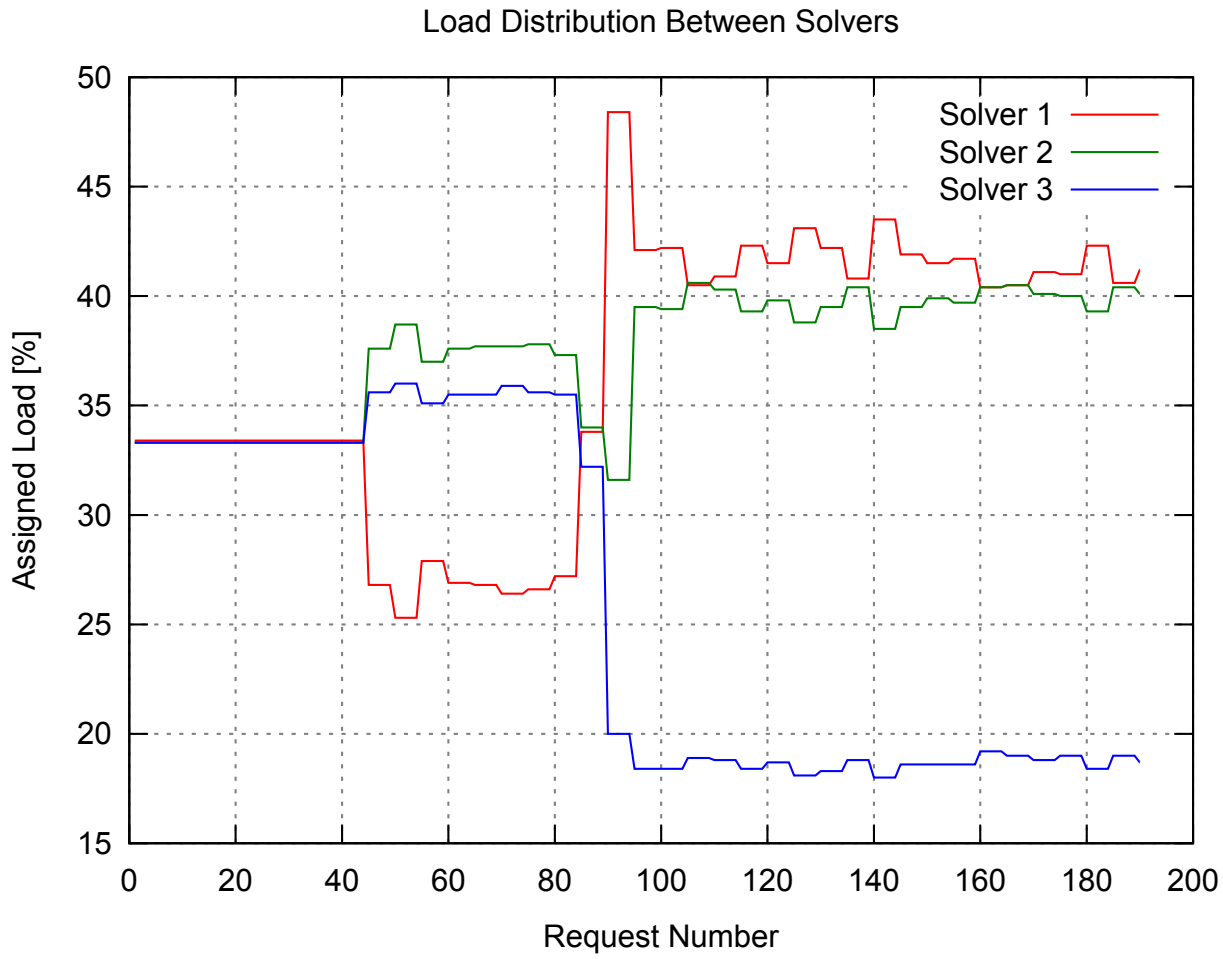

Fig. 5. Load distribution

the modification takes somes iterations to reach a stable situation, and more work is assigned to *Solver 2* and *Solver 3*, and less to *Solver 1* (see Figure 5), reaching a situation in which all solvers take similar response time (Figure 6).

Finally, after the architectural change in Phase 3, *Solver 1* becomes more powerful than the other solvers. The self-balancing system detects a deviation from the last computed *optimalBal*, and recalculates the optimal distribution, leading again to a situation in which all solvers receive different amounts work such that all of them takes similar time to execute. Notice that through each phase, the performance of the application varies until it reaches a stable situation, effectively improving the throughput, and the response time perceived by the client (which is $max(s1, s2, s3)$) decreases.

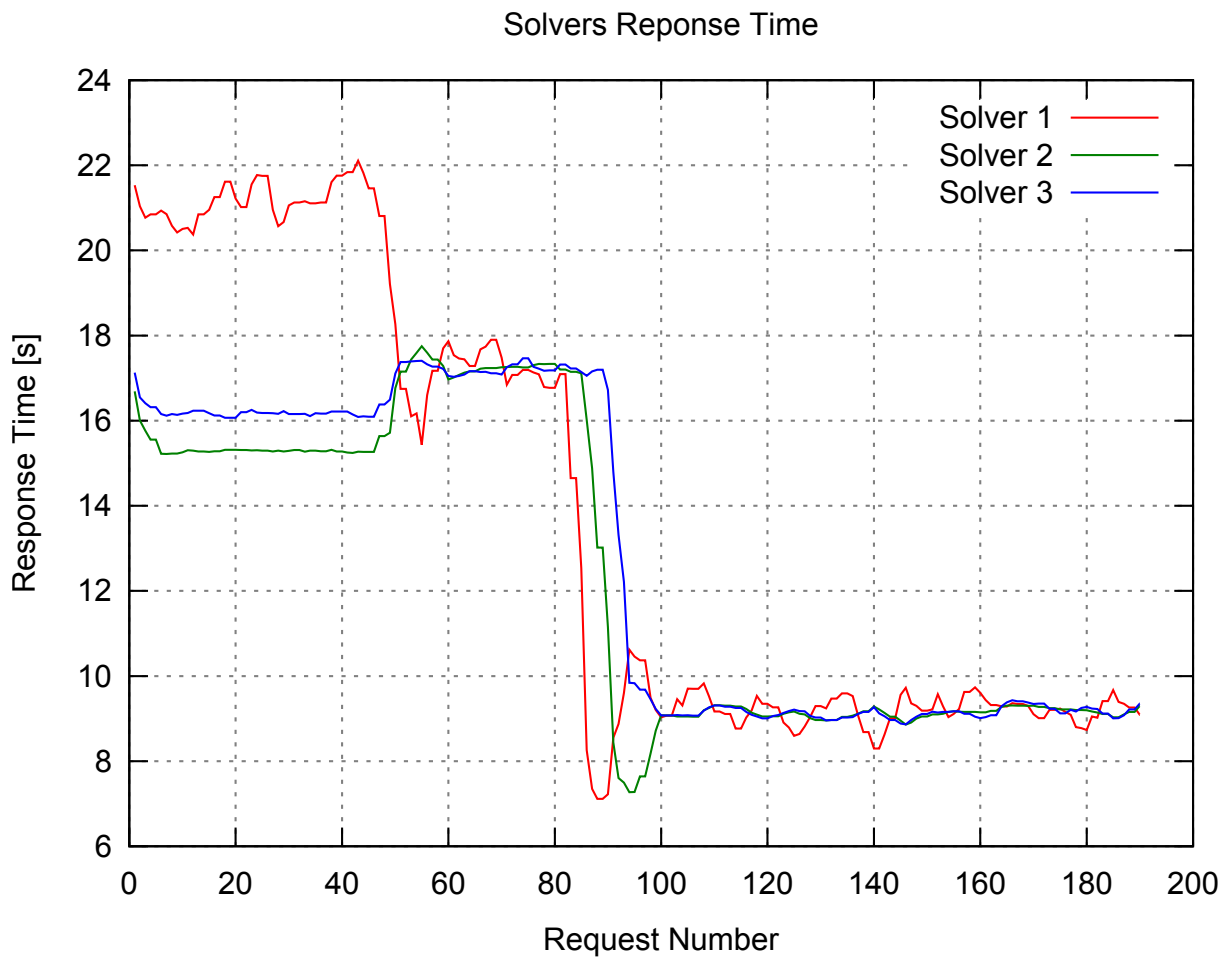

Fig. 6. Solvers response time for the cracking request

6 Conclusions

In this work we have presented an extension to the GCMScript scripting language that allows to easily program effective, dynamic reconfigurations, and can be used to build autonomic behavior. We have shown our contribution by providing self-optimizing capabilities to a component-based GCM/ProActive distributed application. The results show that, through our scripting extensions, little programming effort is required to introduce a behavior that automatically self-adjust the load distributed to several worker nodes in an unmanaged way after a few executions, and it is also able to react to architectural or environmental changes in the infrastructure, which is a possible scenario in cloud supported infrastructures. A similar result would be much harder and slower to achieve by a human manager.

We expect that this model provides a basis for building more complex self-adaptation capabilities, and ease the construction of autonomic cloud applications. Future work considers the development of self-repair capabilities, for example, when a worker component is randomly taken out of line, or deployed in a cloud that suddenly becomes unreachable. Though this is also an architectural change in the application, additional precautions must be taken to avoid possible inconsistencies or perform some necessary rollbacks as part of the recovery process. We expect also to experiment with self-protecting capabilities that consider a prediction process that allows to anticipate a possible degradation either in the application or in the environment.

ACKNOWLEDGEMENTS

This work was partially funded by CIRIC-INRIA Chile, SCALE team at INRIA Sophia-Antipolis and the SCADA associated team.

BIOGRAPHIES

Matías Ibañez is a student of the Computer Science Department at the University of Chile. During the year 2014 he did a trainee at NIC Chile Research Labs and a research visit at INRIA Sophia-Antipolis research center. His research interest include Distributed Systems and Component Models. Contact him at matias@niclabs.cl.

Cristian Ruz is a current researcher at the Computer Science Department at Pontificia Universidad Católica de Chile. His research interests are in the area of distributed systems, in particular in parallel computing and use of HPC architectures, heterogenous systems, and component-based software development. Contact him at cruz@ing.puc.cl.

Ludovic Henrio is the scientific leader of the SCALE team, he has got a position (CR1) at CNRS, in I3S lab. His research interest include Semantics, Object Calculi, Components, Concurrency and Distribution, Confluence and Determinacy, and Distributed Systems. Contact him at ludovic.henrio@cnrs.fr.

Javier Bustos-Jiménez is the head of NIC Chile Research Labs, an institution affiliated to the University of Chile. His research interests include networking, internet protocols, and mobile and distributing computing. Bustos-Jiménez has a Diplome de Docteur d'Informatique from the University of Nice Sophia, France. Contact him at jbustos@niclabs.cl.

REFERENCES

[1] E. Bruneton, T. Coupaye, and J. Stefani, "The fractal component model specification." http://fractal.objectweb.org/specification/index.html, 2004.

[2] F. Baude, L. Henrio, and C. Ruz, "Programming distributed and adaptable autonomous components-the gcm/proactive framework," *Software: Practice and Experience*, pp. n/a–n/a, 2014.

[3] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures," *Annals of Telecommunications*, vol. 64, pp. 45–63, 2009.

[4] L. Henrio and M. Rivera, "Stopping safely hierarchical distributed components," in *Proceedings of the Workshop on Component-Based High Performance Computing (CBHPC'08) in conjunction with ACM SIGPLAN CompArch 2008*, October 2008.

[5] C. Germain-Renaud and O. F. Rana, "The convergence of clouds, grids, and autonomics," *IEEE Internet Computing*, vol. 13, no. 6, p. 9, 2009.

[6] J. Bustos-Jiménez, D. Caromel, M. Leyton, and J. Piquer, "Coupling contracts for deployment on alien grids," in *Euro-Par 2006: Parallel Processing* (W. Lehner, N. Meyer, A. Streit, and C. Stewart, eds.), vol. 4375 of *Lecture Notes in Computer Science*, pp. 61–73, Springer Berlin Heidelberg, 2007.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[8] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 187–197, ACM, 2002.

[9] C. Tibermacine, D. Hoareau, and R. Kadri, "Enforcing architecture and deployment constraints of distributed component-based software," in *Fundamental Approaches to Software Engineering* (M. Dwyer and A. Lopes, eds.), vol. 4422 of *Lecture Notes in Computer Science*, pp. 140–154, Springer Berlin Heidelberg, 2007.

[10] IBM, "An architectural blueprint for autonomic computing.," *white paper*, vol. Fourth Edition, no. June, 2006.

[11] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, 2012.

[12] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pp. 40 –48, aug. 2006.

[13] J. Buisson, F. André, and J.-L. Pazat, "A framework for dynamic adaptation of parallel components," in *Parallel Computing: Current & Future Issues of High-End Computing International Conference ParCo*, vol. 33 of *NIC Series*, (Malaga Spain), p. 65, 2005.

[14] P. Martin, A. Brown, W. Powley, and J. Vazquez-Poletti, "Autonomic management of elastic services in the cloud," in *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pp. 135–140, June 2011.

[15] F. de Oliveira, T. Ledoux, and R. Sharrock, "A framework for the coordination of multiple autonomic managers in cloud environments," in *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pp. 179–188, Sept 2013.